F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

FIDO2 USB Security Key

Martin Endler
Open Informatics – Cybersecurity

May 2025
https://github.com/pokusew/fel-masters-thesis
Supervisor: Ing. Jan Sobotka, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Endler Martin** Personal ID number: **483764**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

FIDO2 USB Security Key

Master’s thesis title in Czech:

FIDO2 USB bezpečnostní klíč

Name and workplace of master’s thesis supervisor:

Ing. Jan Sobotka, Ph.D. Department of Measurement FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **02.09.2024** Deadline for master's thesis submission: **23.05.2025**

Assignment valid until: **15.02.2026**

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student’s signature

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Endler Martin** Personal ID number: **483764**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

FIDO2 USB Security Key

Master’s thesis title in Czech:

FIDO2 USB bezpečnostní klíč

Guidelines:

FIDO2 is a set of standards based on asymmetric cryptography that enables easy, secure, and phishing-resistant authentication.
The goal of this work is to create a new open-source implementation of a FIDO2 USB hardware external authenticator that is well-documented and thoroughly tested and offers a detailed yet accessible insight into the inner workings of FIDO2, which is something that existing implementations currently lack.
1. Make yourself familiar with the FIDO2 set of standards.
2. Review suitable technologies and existing similar projects.
3. Implement a working FIDO2 USB hardware external authenticator (“security key”) from scratch. External libraries can be used for some low-level generic components.
4. Follow software development best practices and use applicable software quality assurance methodologies.
5. Demonstrate the working of the implementation with authentication flows on real WebAuthn-enabled websites.
6. Document the work and make it publicly available on GitHub.

Bibliography / sources:

[1] W3C (April 8, 2021). Web Authentication: An API for accessing Public Key Credentials – Level 2. W3C Recommendation. https://www.w3.org/TR/webauthn-2/
[2] FIDO Alliance (June 21, 2022). Client to Authenticator Protocol (CTAP). CTAP 2.1 Proposed Standard. https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-errata-20220621.html
[3] USB Implementers Forum (April 27, 2000). USB 2.0 Specification. https://usb.org/document-library/usb-20-specification

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Endler Martin
Personal number: 483764
Programme name: Open Informatics

declare that I have elaborated the master's thesis entitled

FIDO2 USB Security Key

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I did not use any artificial intelligence tools during the preparation and writing of my thesis. I am aware of the consequences if manifestly undeclared use of such tools is determined in the elaboration of any part of my thesis.

In Prague on 23.05.2025

Bc. Martin Endler
.....
student's signature

Errata

Version of this text: **2025-06-18 10:36 CEST**

The latest version of this text can be found at the following URL:
`https://github.com/pokusew/fel-masters-thesis`

Acknowledgement

First, I would like to thank my supervisor Ing. Jan Sobotka, Ph.D. for his guidance, support, and patience. Second, I would like to thank my family and close friends for always supporting me throughout my studies.

Abstract / Abstrakt

FIDO2 is a set of standards based on asymmetric cryptography that enables easy, secure, and phishing-resistant authentication. The strong support from Google, Microsoft, Apple, and other major technology companies in the FIDO Alliance is driving the adoption of FIDO2 across the industry.

This thesis focuses on creating a new open-source FIDO2 USB hardware external authenticator. We provide an overview of the relevant technologies and standards, specifically Web Authentication and CTAP2. We also review existing similar projects. Then, we proceed to the actual implementation. We document our decisions and the most important parts. The resulting fully functional CTAP 2.1 compliant implementation is publicly available on GitHub. It utilizes hardware-accelerated cryptography on the STM32H533 MCU, passes FIDO2 conformance tests, and is usable for authentication on real WebAuthn-enabled websites.

We hope our work will contribute to the popularization and better understanding of the FIDO2 technology.

Keywords: FIDO2, WebAuthn, CTAP, CTAP2, USB, CTAPHID, security key, passkeys, authenticator, passwordless authentication, multi-factor authentication, MFA, second-factor authentication, 2FA, asymmetric cryptography, public key cryptography, Elliptic-curve cryptography, ECC

FIDO2 je sada standardů založených na asymetrické kryptografii, která umožňuje snadnou, bezpečnou a proti phishingu odolnou autentizaci. Silná podpora ze strany společností Google, Microsoft, Apple a dalších významných technologických firem v rámci FIDO Alliance podporuje adopci FIDO2 v celém odvětví.

Tato práce se zaměřuje na vytvoření nového open-source FIDO2 USB hardwarového externího autentizátoru. Nejdříve popisuje relevantní technologie a standardy, zejména Web Authentication a CTAP2. Také zmiňuje existující podobné projekty. Poté přistupuje k samotné implementaci. Dokumentuje rozhodnutí a nejdůležitější části. Výsledná plně funkční implementace splňující CTAP 2.1 specifikaci je veřejně dostupná na GitHubu. Využívá hardwarově akcelerovanou kryptografii na STM32H533, prochází testy shody FIDO2 a je použitelná pro autentizaci na skutečných webových stránkách s podporou WebAuthn.

Doufáme, že naše práce přispěje k popularizaci a lepšímu porozumění technologii FIDO2.

Klíčová slova: FIDO2, WebAuthn, CTAP, CTAP2, USB, CTAPHID, bezpečnostní klíč, passkeys, autentizátor, přihlášení bez hesla, vícefaktorová autentizace, MFA, dvoufaktorová autentizace, 2FA, asymetrická kryptografie, kryptografie s veřejným klíčem, kryptografie nad eliptickými křivkami, ECC

Překlad titulu:
FIDO2 USB bezpečnostní klíč

Contents /

/ Figures

Chapter 1
Introduction

Passwords as a means of authentication suffer from many problems. More than 80% of confirmed breaches are related to stolen, **weak, or reused passwords** [1]. Password-based credentials are the target of **phishing attacks**, which are becoming more sophisticated every day. This poses a major threat since 84% of users reuse the same passwords across multiple sites [2]. To limit this threat and to increase the overall security of authentication, two-factor (2FA) or multi-factor (MFA) authentication flows are being increasingly used [2].

However, the standard MFA mechanisms, including one-time codes delivered via insecure channels (such as SMS, voice call, or email), TOTPs (e.g., Google Authenticator), and proprietary push notification-based systems, do not provide sufficient security. Not only are they all still susceptible to phishing attacks, but they also greatly hinder the user experience.

To solve this problem, the FIDO Alliance was launched in 2013. It develops and promotes strong authentication standards that "help reduce the world's over-reliance on passwords" [3].

Its latest set of standards, jointly developed with the W3C (World Wide Web Consortium), is called **FIDO2**. It is based on asymmetric cryptography, and it enables **easy, secure, and phishing-resistant authentication** for online services (primarily on the web, but it can be used in native applications as well). It supports *passwordless*, *second-factor*, and *multi-factor* user experiences with *platform authenticators* (such as Apple ID with Face ID or Touch ID, Windows Hello, and Google Password Manager on Android) or external *(roaming) authenticators* (such as **FIDO2 security keys**). All major OSs, browsers, and a growing number of websites and applications support FIDO2 [4–6].

Among the FIDO Alliance's 250 members are all the major technological companies [7]. Google (one of the FIDO founding members), Microsoft, and Apple have become vocal advocates for passwordless authentication based on passkeys (the end-user-centric term for FIDO2 credentials) since 2022 when they announced their public commitment to expand support of FIDO2 [8].

With the ever-increasing adoption of FIDO2 across the industry [1], it is useful to understand how this technology works and what its benefits are. In order to do that, we decided to **create a new open-source implementation of an external (roaming) FIDO2 authenticator from scratch**.

1.1 Structure

This thesis is structured as follows:
First, in Section 1.2, we define **the objective** of the thesis.
Second, in Chapter 2, we establish the necessary theoretical foundation by **describing FIDO2** in detail. We also review existing projects related to our goal in Chapter 3.
Next, in Chapter 4, we describe **the actual implementation**, which represents the main contribution of this thesis. Then, we **test and evaluate** our implementation in Chapter 5.
Finally, in Chapter 6, we conclude the thesis by summarizing **the achieved results**.

1.2 Thesis Objective

The objective of this work is to create a new **open-source implementation of a FIDO2 USB hardware-based security key** that is fully functional, well-documented and thoroughly tested and offers a detailed yet accessible insight into the inner workings of FIDO2, which is something that existing open-source implementations currently lack.

In terms of WebAuthn (which we explain in Chapter 2), we aim to create a roaming authenticator with cross-platform attachment using CTAP 2.1 over USB 2.0 (CTAPHID) as the communication protocol, that supports user verification using PIN (CTAP2 ClientPIN), and is capable of storing passkeys (client-side discoverable credentials).

The end result should be similar to proprietary commercial products such as those from Yubico 1.1 or GoTrust 1.2. Nevertheless, our work focuses on the software implementation (optimized for use on resource-constrained hardware platforms), not on the physical product or the hardware design. As a part of our work, we will select a suitable MCU-based hardware platform for our implementation.

We want to create an implementation that is self-contained, with a minimum number of dependencies. By implementing all the key parts from scratch, we can develop a deep understanding of FIDO2/WebAuthn.

Finally, by focusing on the quality and the documentation, this implementation could help others understand the FIDO2 standards, and, in general, contribute to the popularization of FIDO2 technology.

Figure 1.1. Security Key NFC by Yubico, one of the most typical FIDO2 USB/NFC security keys (or more precisely, cross-platform roaming authenticators).

Figure 1.2. GoTrust Idem Key, a FIDO2 USB/NFC security key with FIDO Authenticator Certification Level 2 (L2) [9].

Chapter 2
FIDO2

In this chapter, we provide a more detailed description of **FIDO2**.

FIDO2 is a set of related specifications, jointly developed by the FIDO Alliance and the W3C (World Wide Web Consortium), that together enable **easy, secure, and phishing-resistant authentication** for online services (primarily on the web, but they can be used in native applications as well).

The specifications are:

- **Web Authentication (WebAuthn) API** by World Wide Web Consortium (W3C)

  This is the core specification that **defines and describes all the key concepts**, some of which we cover in the following sections. More specifically, it "defines an API enabling the creation and use of strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users" [10].

  Level 2 [10] (W3C Recommendation from April 8, 2021) is the latest stable version.

  Level 3 [11] is being actively developed and some of its new features (hybrid transport, conditional mediation, hints, credentials backup) are already supported by browsers and authenticators.

- **Client to Authenticator Protocol (CTAP)** by FIDO Alliance

  "This specification describes an application layer protocol for **communication between a roaming authenticator and another client**/platform, as well as bindings of this application protocol to different transport protocols" [12].

  CTAP 2.1 [12] (Proposed Standard from June 21, 2022) is the previous version.

  CTAP 2.2 [13] (Proposed Standard from February 28, 2025) is the latest stable version. The main difference compared to CTAP 2.1 is the addition of hybrid transports. Apart from that and a few new extensions, it is completely the same as CTAP 2.1 (There is no new version identifier for CTAP 2.2, authenticators still use `FIDO_2_1`). Because none of those changes are relevant for our work, we will refer to CTAP 2.1 for the rest of our text.

- There is also FIDO U2F (now referred to as CTAP1 or CTAP1/U2F), which is a predecessor of FIDO2 that can be used only for two-factor authentication (as a second factor). FIDO2 authenticators can be backwards-compatible with CTAP1 (FIDO U2F). To avoid unnecessary complexity, we will not add support for CTAP1/U2F to our implementation. Therefore, we will not describe it.

2.1 Public Key Cryptography

FIDO2 relies on public key cryptography (also called asymmetric cryptography) [10].

Public key cryptography uses the concept of a key pair. Each key pair consists of a **private key** (which must be kept secret) and a corresponding **public key** (which can be openly distributed without compromising security). The public and private keys are mathematically related. They are generated with cryptographic algorithms based on mathematical problems termed *one-way functions* [14]. It is not possible to calculate the private key from the public key. One of the most common public key cryptosystems is RSA that relies on the difficulty of factoring the product of two large prime numbers [15]. Another widely used public key cryptosystem is Elliptic Curve Cryptography (ECC), which is based the algebraic structure of elliptic curves over finite fields, specifically the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP) [16].

2.2 Functional Description

The core idea is that FIDO2/WebAuthn allows servers (i.e., websites, referred to as **Relying Parties**, **RP**) to register and authenticate users using public key cryptography.

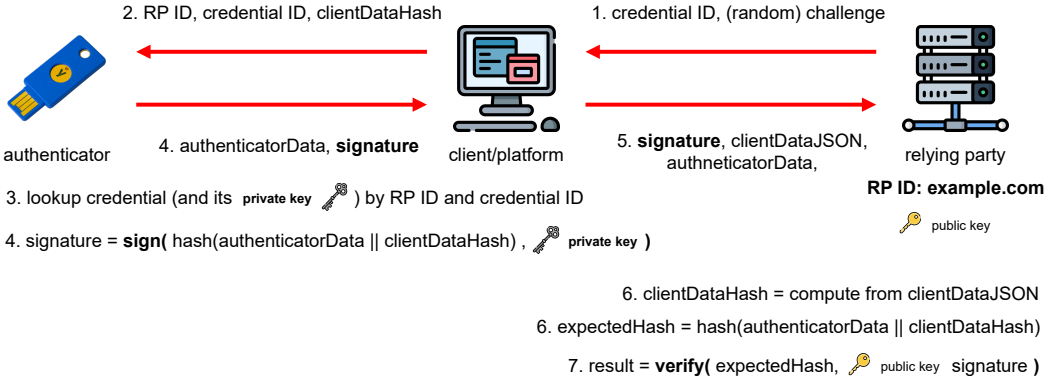

Figure 2.1. Simplified WebAuthn authentication flow. The credentials belong to the user and are managed by an authenticator, with which the Relying Party interacts through the client platform.

Instead of a password, a private-public keypair (known as a **credential**) is created for a website during registration. While the **private key** is stored securely on the user's device (inside the **authenticator**) and it never leaves it, the **public key** and an authenticator-generated credential ID are sent to the server where they are stored.

Then, during **authentication**, the RP's server then uses that **public key** to verify the user's identity by verifying the user's possession of the corresponding private key. More specifically, the server sends the credential ID and a random data called a challenge to the client (the user's browser). The client bounds the challenge to the RP ID by hashing the challenge together with the RP ID and producing clientDataHash. The client passes RP ID, credential ID, and clientDataHash to the user's authenticator. The authenticator looks up the **private key** by the RP ID and credential ID and it signs the concatenation of authenticatorData (which contains info about the authenticator) and clientDataHash using the **private key**. It passes this signature back to the server via the client. The server checks that all values in the clientDataJSON and in the authenticatorData have the expected values (especially RP ID). Then it computes the

expected hash using the clientDataJSON and authenticatorData. Finally, it uses the **public key** to verify the signature.

The exact signature algorithm depends on the supported algorithms by the RP and by the authenticator. WebAuthn does not list the specific algorithms, instead it refers to the IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry[1]. One of the common algorithm identifiers is `-7`, which stands for Elliptic Curve Digital Signature Algorithm (ECDSA) on the P-256 curve (also called secp256r1 or prime256v1) with SHA-256 as the hashing function.

2.3 Key Benefits

Based on the description in the previous section, we can clearly see some of the security and privacy benefits of FIDO2 authentication.

- **Security**
  - A new unique public-private key pair (credential) is created for every relying party (and every authenticator the user uses).
  - All credentials are scoped to the relying party (website).
  - The private key never leaves the authenticator.
  - The RP's server does not store any secrets.
  - The possession of the private key is proved by signing a random challenge generated by the RP.
  - This security model **eliminates all risks of phishing, all forms of password theft** (there is **no shared secret to steal**) and **replay attacks** (thanks to the random challenge).
- **Privacy**
  - The whole FIDO2 is designed with privacy in mind. All credentials are unique and they are scoped to the relying party (website). They cannot be used to track users in any way.
  - Plus, biometric data, when used for user verification, never leaves the user's device (authenticator).
- **User Experience**
  - Users interact with the authenticator by (typically) simply touching it to provide consent with registration and authentication.
  - Credentials can be further protected and their use might require local user verification, which, based on the authenticator capabilities, might utilize PIN or biometric methods.
  - Users can use cross-platform *roaming authenticators* such as FIDO2 security keys (which communicate via USB, NFC, Bluetooth) or they might rely on their-device built-in *platform authenticators* (such as Apple ID with Face ID or Touch ID, Windows Hello, and Google Password Manager on Android) that are part of the client device.

2.4 Terminology

In the previous sections, we briefly introduced the key parties involved in FIDO2 – *the relying party (RP), the client, and the authenticator.* In this section, we provide more detailed information about them and their behavior. We also define additional related terms to aid in our explanations.

[1] `https://www.iana.org/assignments/cose/cose.xhtml#algorithms`

2.4.1 Relying party (RP)

"An entity whose application utilizes the WebAuthn API to register and authenticate users, and which stores the public key." [10] Typically, such an application has both client-side code to invoke the WebAuthn API on the client (typically in the browser) and server-side code to validate responses and store details about registered credentials (their IDs and public keys) in some sort of database.

2.4.2 Client

An entity (typically a web browser or a similar application) that acts as an intermediary between the relying party and the authenticator [10].

2.4.3 Client Device

"The hardware device on which the client runs, for example a smartphone, a laptop computer or a desktop computer, and the operating system running on that hardware." [10]

2.4.4 Client Platform

"A client device and a client together make up a client platform. A single hardware device may be part of multiple distinct client platforms at different times by running different operating systems and/or clients." [10]

2.4.5 Authenticator

"A cryptographic entity, existing in hardware or software, that can register a user with a given Relying Party" (i.e., generate and store a public-private key pair with appropriate metadata) and "later assert possession of the private key during authentication". [10] The private keys never leave the authenticator.

The authenticators that are part of the client device are referred to as *platform authenticators* (such as Apple ID with Face ID or Touch ID, Windows Hello, and Google Password Manager on Android), while those that are reachable via cross-platform transport protocols (USB, NFC, Bluetooth) are referred to as *roaming authenticators* (such as **FIDO2 USB security keys**).

2.4.6 AAGUID

Authenticator Attestation Globally Unique Identifier. A 128-bit identifier indicating the type (e.g. make and model) of the authenticator, chosen (randomly generated) by its manufacturer.

2.4.7 Credential

A public-private key pair that is used for authentication. It is bound to an RP ID. A credential is identified by a **Credential ID**, which is an opaque probabilistically-unique byte string generated by authenticator that can be up to 1023 bytes long. It is associated with a user account, which is identified by a **user handle** (`user.id`). A user handle is an opaque byte sequence with a maximum size of 64 bytes, generated by the RP and stored together with the credential by the authenticator. Optionally, the user account information within the credential can also contain `name` and `displayName`, which are both arbitrary-length strings. Every credential is either discoverable (also called client-side discoverable) or non-discoverable. RP controls which type of credential will be created during registration by passing an appropriate option to the WebAuthn API.

Discoverable credentials are also called **"passkeys"** [11]. RP pass credential IDs during registration (in the excludeList) to prevent users from creating multiple credentials for the same user account on a single authenticator. RP can also pass credential IDs during authentication (in the allowList) to explicitly allow authentication (i.e., generating an assertion, resp. a signature) only using those credentials. When a credential is non-discoverable, its Credential ID must be passed to authenticator during each authentication call.

2.4.8 User Verification

"The process by which an authenticator locally authorizes the invocation of an operation. Various authorization gesture modalities are possible, for example, a touch plus pin code, password entry, or biometric recognition (e.g., presenting a fingerprint). The intent is to distinguish individual users." [10].

The "user verification does not give the RP a concrete identification of the user, but when two or more ceremonies with user verification have been done, it expresses that it was the same user that performed all of them. The same user might not always be the same natural person, however, if multiple natural persons share access to the same authenticator." [10].

2.4.9 User Presence

"A test of user presence is a simple form of authorization gesture and technical process where a user interacts with an authenticator by (typically) simply touching it." [10].

2.4.10 Credential Storage Modality

Authenticators (such as FIDO2 hardware security keys) might have a limited storage capacity. Until now, we have assumed that **the private keys** must be stored in the authenticator. In fact, the WebAuthn allows **two different storage strategies**:

1. The private key is stored **in persistent storage** embedded in the authenticator. This strategy is necessary for **discoverable credentials** (also called **client-side discoverable credentials** or **passkeys**) that can be used as a first factor.
2. The private key is **stored within the credential ID**. Specifically, the private key is encrypted **(wrapped)** such that only the authenticator can decrypt (i.e., unwrap) it. The resulting ciphertext is **the credential ID**.

   Typically, the authenticator uses symmetric AES encryption, where the symmetric encrypt/decrypt key (sometimes called master key) is securely stored in the authenticator and never leaves it. Therefore, this strategy, if correctly implemented, is as secure as the first one.

   Note that this strategy is possible because **the RP passes the credential ID** to the authenticator **during authentication**. This is the case for the non-discoverable credentials. Therefore, the authenticator does not need to store any information about such a credential in its persistent storage. This way it can support virtually an unlimited number of credentials. However, this strategy **cannot** be used for discoverable credentials, where the RP does **not** pass the credential ID to the authenticator and instead the authenticator must be able to list all existing credentials for that RP.

Note that the authenticators **might support both strategies** and use different storage strategies for different credentials (typically only use the persistent storage when it is strictly necessary, i.e., for *discoverable* credentials).

For more information, see 6.2.2. Credential Storage Modality in [10].

2.5 Ceremonies

In this section, we build on the information provided in the Functional Description section, and we cover the **registration** and **authentication** flows in more detail. These flows are referred to as "ceremonies" by the WebAuthn specification because they extend the concept of a computer *communication protocol* with human-computer interactions.

In the following descriptions, we assume **a standard web application** that runs client-side code to invoke the WebAuthn JavaScript APIs in the browser and to communicate with its server, which validates responses and stores/retrieves details about registered credentials (their IDs and public keys). In this case, the browser represents the WebAuthn client. Note that WebAuthn can also be used in native applications where platform-specific vendor-provided APIs are used in place of the WebAuthn JavaScript APIs. On Android, these are FIDO2 API for Android[2] and/or Credential Manager API[3]. iOS provides similar APIs[4].

2.5.1 Registration

During registration, a new credential is created on an authenticator and registered with a Relying Party server. Registration must happen before a user can use their authenticator to authenticate.

The following diagram 2.2 from depicts the registration flow:

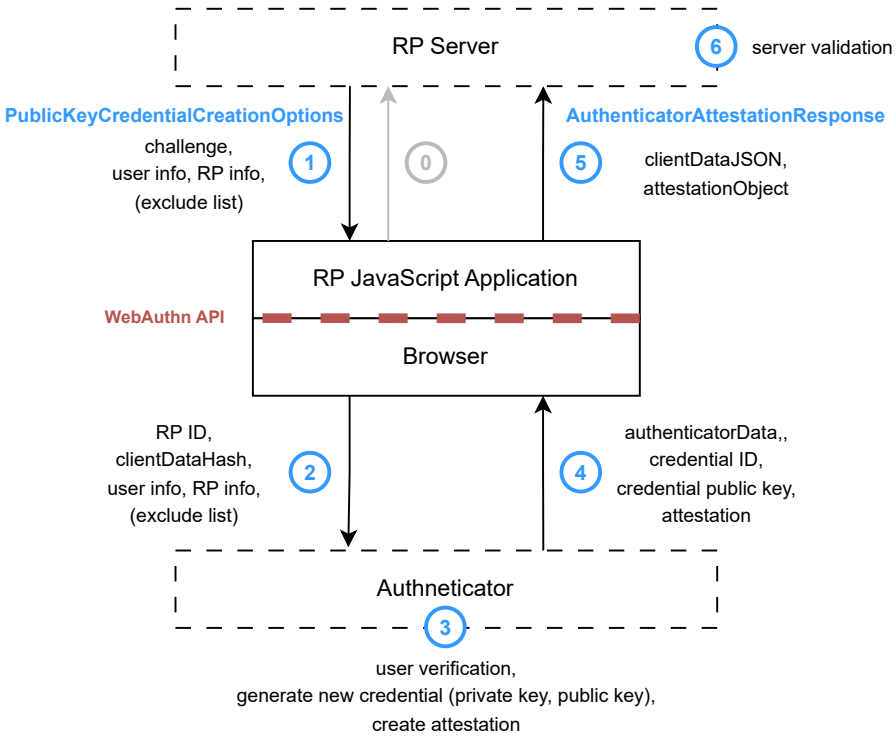

Figure 2.2. The WebAuthn registration flow.

The user visits a website, for example, `example.com`, which serves up a script. At this point, the user may already be logged in using a legacy username and password, or

[2] `https://developers.google.com/identity/fido/android/native-apps`
[3] `https://developers.google.com/identity/android-credential-manager`
[4] `https://developer.apple.com/documentation/authenticationservices/public-private-key-authentication`

additional authenticator, or other means acceptable to the Relying Party. Or the user may be in the process of creating a new account.

0. The user initiates the registration process by interacting with the application in the browser. The application sends a request to the server to start the registration.
1. The server responds with `PublicKeyCredentialCreationOptions` that contains information about the user (the opaque *user handle*, also called `user.id`), preferred authenticators (roaming vs. platform) and *supported algorithms*. It also includes a random data called **challenge**. This is to prevent replay attacks.
2. The client-side code in the browser invokes the WebAuthn API method `navigator.credentials.create()` and passes the options from the server. The browser **appends the RP ID** (the website origin, i.e., example.com). This is necessary for the **correct scoping** of the credentials to the RP (every credential is *bound* to an RP ID). Also, the browser appends **clientDataHash**, which is a hash over the operation type (`webauthn.create` or `webauthn.get`), the challenge, and the origin. Then, the browser locates a suitable authenticator, establishes a connection to it, and sends the `authenticatorMakeCredential` request. The communication between the browser (the client) and the authenticator uses CTAP2 in case of roaming authenticators (e.g., a FIDO USB security key) or some platform-specific communication channel in case of platform authenticators (e.g. Windows Hello).
3. The authenticator asks the user for some sort of authorization gesture to provide consent. It may involve user verification (PIN entry or biometric check) or it may only be a simple test of user presence. If the user authorizes the registration, the authenticator **generates a new credential** (a public-private key pair and a credential ID). The private key is securely stored in the authenticator and never leaves it.
4. The authenticator takes **the generated public key** and **the credential ID**, appends the clientDataHash and information about itself and signs all this data with either its attestation private key or with the credential private key (so called self-attestation).
5. The client-side code in the browser obtains the response from the authenticator (as the result of the `navigator.credentials.create()` call) and sends it to the server.
6. The server verifies the data. Specifically, it checks the attestation signature. This way, the server ensures that all the credential creation options are respected and that the response is indeed related to the initial request challenge. If everything matches, **the server saves the public key and the credential ID to the database** and associates them with the user.

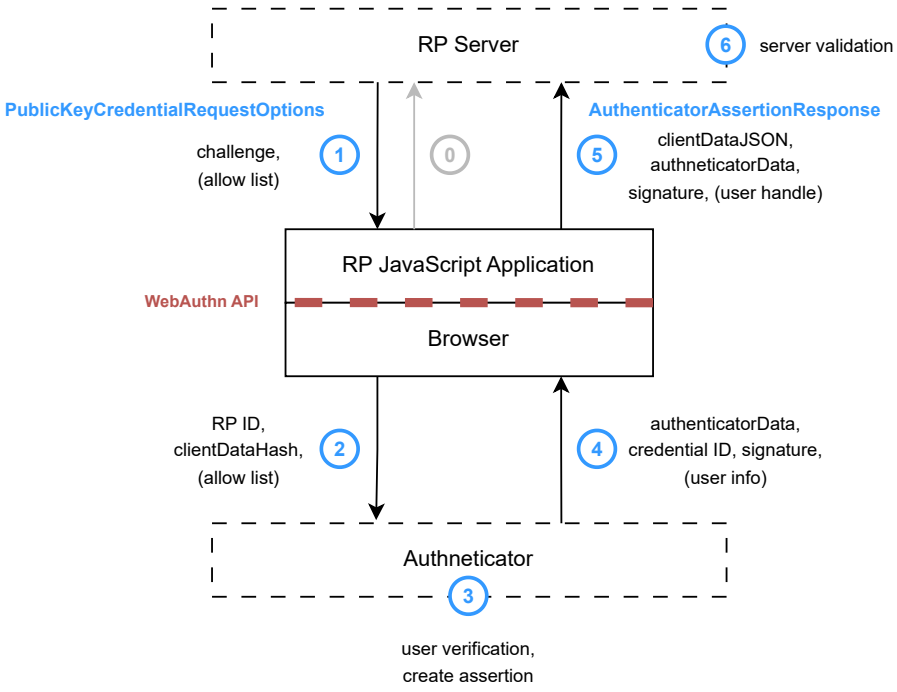

Figure 2.3. The WebAuthn authentication flow.

2.5.2 Authentication

This is the ceremony when a user with an already registered credential visits a website and wants to authenticate using the credential (`authenticatorGetAssertion` request). During authentication, the server uses the previously registered public key to verify the user's identity. Specifically, it verifies the user's possession of the private key by validating the authenticator-generated assertion signature. The complete authentication flow is depicted in Figure 2.3.

Since we have already described the authentication in sufficient detail in the Functional Description section, we omit the detailed description here and ask the reader to refer to the corresponding description there.

2.6 CTAP2

WebAuthn defines the key concepts and principles of the whole FIDO2 functionality. However, it does not define the communication protocol between the client and the authenticator. It introduces an abstract functional model for a WebAuthn Authenticator. This model defines the necessary observable behavior. Specifically, it defines the authenticator data structure, signature formats (attestations and assertions), and the authenticatorMakeCredential, authenticatorGetAssertion, and authenticatorCancel operations using an abstract notion of connection and session between the client and the authenticator.

CTAP2 represents a concrete instantiation of this model. It defines the actual communication protocol and its bindings to various transport protocols. The specification defines multiple abstraction levels.

2.6.1 Authenticator API

At the highest level of abstraction, CTAP2 defines a conceptual API consisting of operations (also called commands or requests). Each command accepts input parameters

and returns either a response with data on success or an error code. The returned response depends on the parameters and the authenticator's internal state. However, there are also special commands called stateful commands, which do not accept any parameters but require an additional command state to be maintained. "Each such command uses and updates a state that is initialized by a corresponding state initializing command." [12]

Here is a list of all mandatory CTAP 2.1 commands with their command codes. Note that some of the commands have also subcommands (technically, the subcommand code and the subcommand parameters are parameters of the parent command).

- **authenticatorGetInfo (0x04)** – "Using this method, clients can request that the authenticator report a list of its supported protocol versions and extensions, its AAGUID, and other aspects of its overall capabilities. Clients should use this information to tailor their command parameters choices." [12]
- **authenticatorClientPIN (0x06)** – See PIN/UV Auth Protocol
- **authenticatorMakeCredential (0x01)** – "request generation of a new credential in the authenticator" [12]
- **authenticatorGetAssertion (0x02)** – "request cryptographic proof of user authentication as well as user consent to a given transaction, using a previously generated credential that is bound to the authenticator and RP ID" [12]
- **authenticatorGetNextAssertion (0x08)** – (stateful command) "The client calls this method when the authenticatorGetAssertion response contains the numberOfCredentials member and the number of credentials exceeds 1." [12]
- **authenticatorCredentialManagement (0x0A)** – "This command is used by clients to manage discoverable credentials on the authenticator." [12]
- **authenticatorSelection (0x0B)** – "This command allows the client to let a user select a certain authenticator by asking for user presence." [12]
- **authenticatorReset (0x07)** – "Invalidates all generated credentials, erases all discoverable credentials, resets PIN" [12]

2.6.2 PIN/UV Auth Protocol

CTAP2 offers two different ways to perform user verification:

- **ClientPIN** – The entry of the PIN is mediated by the client. The user enters the PIN into the client's UI (e.g., a browser dialog), and the client sends it encrypted to the authenticator. ClientPIN is useful when the authenticator does not have its input UI (e.g., it is a security key without any display or keypad).
- **Built-in User Verification** – User verification is performed entirely within the authentication boundary, e.g., the authenticator has a fingerprint sensor or a secure built-in input UI (e.g., built-in keypad).

The PIN/UV auth protocol (aka pinUvAuthProtocol) handles both ways of user verification. The ClientPINs are encrypted when sent to an authenticator and exchanged for a pinUvAuthToken to authenticate subsequent commands. "Authenticators supporting built-in user verification methods can provide a pinUvAuthToken (again, to authenticate subsequent commands) upon performing the built-in user verification. The pinUvAuthToken is a randomly-generated, opaque bytestring that is large enough to be effectively unguessable." [12] Both PIN and pinUvAuthToken are never sent in plaintext. Instead, they are encrypted using a shared secret that is obtained by performing an ECDH key agreement. However, the plain ECDH key agreement (as is any unauthenticated DH key agreement) is vulnerable to MitM attacks (e.g., an active attacker with the ability to modify USB packets).

The authenticatorClientPIN (0x06) command, resp. its eight subcommands, implement the necessary operations to get the number of remaining PIN/UV retries, obtain a shared secret (using ECDH), set a PIN, change existing PIN, exchange PIN for a pinUvAuthToken, and perform built-in user-verification to get a pinUvAuthToken.

Furthermore, PIN/UV Auth Protocol provides an internal API, which is used by the rest of the Authenticator API commands to validate the pinUvAuthParam. The pinUvAuthParam is used to convey the user verification status while also authenticating the parameters of commands such as authenticatorMakeCredential, authenticatorGetAssertion, and authenticatorCredentialManagement. The client uses the previously obtained pinUvAuthToken to compute an HMAC over all or some of the command parameters. The authenticator verifies the HMAC by computing it from the relevant received parameters and the current pinUvAuthToken (if in use). A successful verification indicates that a user verification performed in the recent authneticatorClientPIN command can now be used to perform the requested operation (makeCredential, getAssertion, or credentialManagement).

2.6.3 Message Encoding

CTAP2 encodes all command parameters and response data using Concise Binary Object Representation (CBOR). This encoding was chosen because many transports (for example, NFC) "are bandwidth-constrained, and serialization formats such as JSON are too heavy-weight for such environments" [12]. Furthermore, CTAP2 introduces the CTAP2 canonical CBOR encoding form, which further restricts the CBOR format with additional encoding rules (such as NO indefinite-length arrays and maps, integers MUST be encoded as small as possible, etc.).

Each operation in the conceptual Authenticator API has a command code (1 byte) assigned. The request message is constructed by prepending the command code to the CBOR-encoded parameters. Analogically, the response message is constructed by

prepending the error code (0 for success) to the CBOR-encoded response data. Since error responses do not have any data, they are only one byte (the error code).

2.6.4 CTAPHID

One of the defined transport protocols for CTAP2 is USB, resp. USB HID. The protocol is called CTAPHID, and it implements logical channel multiplexing so that multiple clients (browsers, apps, OS) on the same client device can communicate with the authenticator concurrently. Next, it introduced the concept of a transaction. The CTAPHID transaction consists of a CTAPHID request followed by a CTAPHID response message. The transactions are always initiated by clients. Request and response messages are divided into individual fragments, known as packets. Packets are directly mapped onto USB HID reports. The fragmentation and channel multiplexing add some overhead to the raw CTAP2 messages (11% – 8% based on the payload length when the HID report size is 64 bytes).

The USB HID report descriptor plays a key role in the automatic *device discovery*. The CTAPHID protocol is designed with the objective of driver-less installation on all major host platforms. Browsers and other clients use the standard USB HID driver included within the OS. The HID *Usage Page* field within the authenticator's HID report descriptor is set to the value `0xF1D0`, which is registered to the FIDO Alliance (see HID Usage Tables 1.6 [17]).

Chapter 3
Existing Work

Before we started working on our own implementation, we did an extensive search to see what open-source implementations of FIDO2 authenticators were available. The most relevant projects are SoloKeys and OpenSK.

3.1 Open-Source Software and Hardware

3.1.1 SoloKeys

- "the first open source FIDO2 security key"
- v1 launched on Kickstarter in 2018 and v2 in 2021.
- As of today (June 2, 2024), it seems that the project development has stopped.
- v1[1] software written in C for STM32L432.
- v2 software written in Rust for NXP LPC55S69.

3.1.2 OpenSK by Google

- An open-source implementation for security keys written in Rust that supports both FIDO2 and FIDO U2F.
- Actively developed by Google[2].
- Written in Rust, uses Tock OS, the main development target platform is Nordic nRF52840.

3.1.3 Nitrokey

- Open Source IT-Security Hardware from a German company.
- Partially based on the SoloKeys project.

3.2 Other Relevant Projects

Two bachelor theses [18–19] at CTU in Prague (both from CTU FIT).

[1] https://github.com/solokeys/solo1
[2] https://github.com/google/OpenSK

Chapter 4
Implementation

This chapter describes the main parts of our FIDO2 authenticator implementation.

The complete source code and additional documentation is available on GitHub:
`https://github.com/pokusew/lionkey`

4.1 Goals

- self-contained implementation, with **a minimum number of dependencies**
- understandable, clear implementation of the standard
- hardware-independent core, making the whole project easily portable
- OS-independent, capable of running as a bare-metal application on the MCU
- testable
- hardware-accelerated cryptography

4.2 Decisions

We decided to write our implementation in **C**, which is a very popular language for embedded development. It allows easy access to the underlying hardware (peripheral registers). We did consider using a more high-level language such as Rust, but we wanted to focus primarily on the CTAP2 standard implementation, rather than learning a new language. Moreover, there is already a high-quality CTAP2 implementation in Rust within the OpenSK project.

In our implementation, we avoided the use of dynamic memory allocations. All memory is allocated statically or on the stack. We also avoided the use of global variables. All functions only modify the state that they are given. For some parts of the implementation, we also adopted object-oriented techniques to improve testability and enable more flexibility.

4.3 Maintaining Quality

Implementing any larger software product requires rigorous quality assurance practices to ensure security, reliability, and maintainability.

We implemented unit tests that allow testing the core of the implementation directly on the host system without the need for the hardware. We tracked code coverage and strived to write better tests. We also configured sanitizers (AddressSanitizer, UndefinedBehaviorSanitizer) to help prevent additional class of possible bugs.

We also set up CI/CD pipeline on GitHub Actions.

Additionally, we relied on the static checks (Clang-Tidy and CLion's advanced static code and flow analyzers). Some of those could be integrated into the CI/CD pipeline as well.

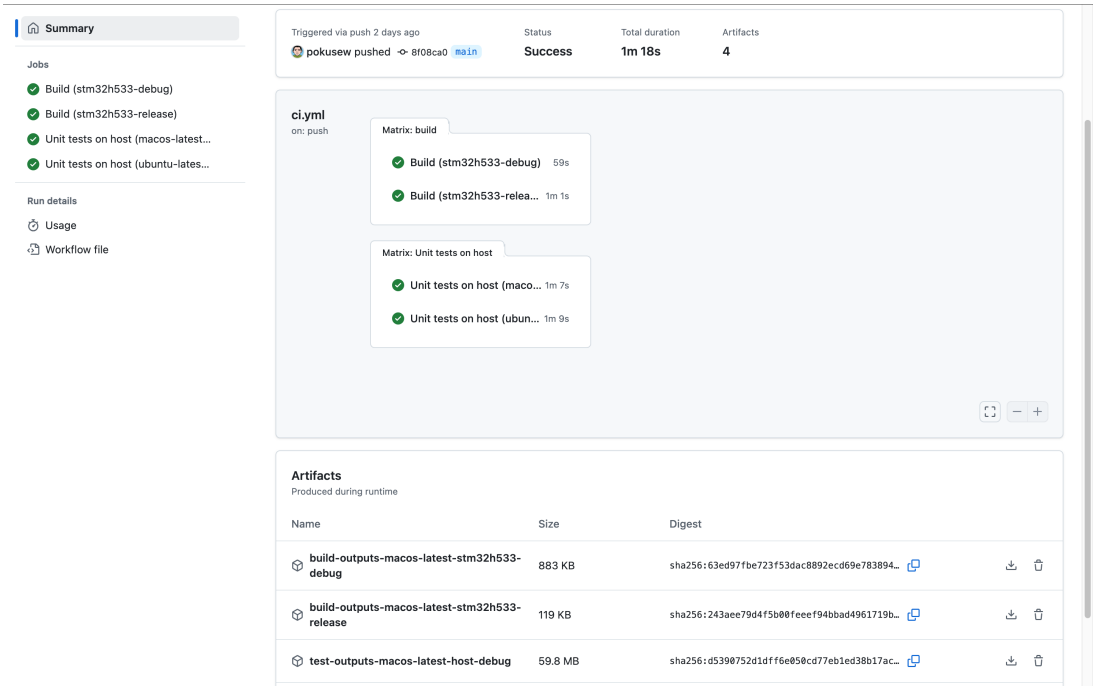

Figure 4.1. A recent run of our CI/CD pipeline in GitHub Actions.

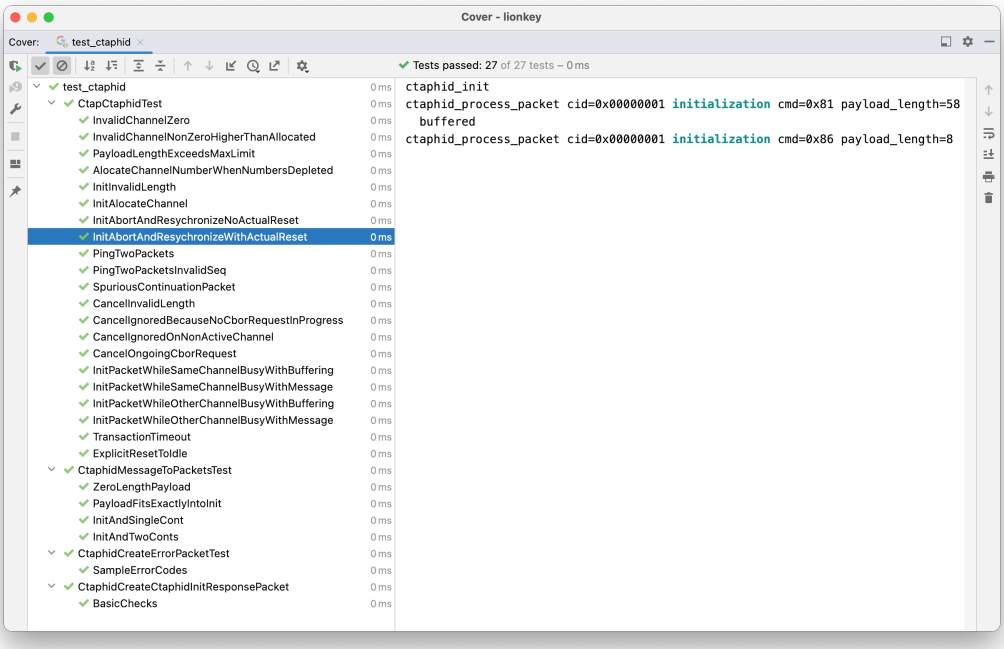

Figure 4.2. The CTAPHID test suite. 100% lines coverage (not visible in the screenshot).

4.4 Choosing Hardware

We sought a hardware platform that would meet the needs of our implementation while being accessible and usable by the builder community worldwide. Therefore, we established the following criteria:

- Available and affordable development kits (boards) with integrated hardware debugger and programmer.
- Freely available and high-quality documentation (Reference Manual, Programming Manual, etc.). No NDA is needed.
- No legacy or deprecated MCUs. Only active products.
- USB 2.0 FS peripheral.
- Single-core 32-bit MCU, such as the Arm Cortex-M line.
- Reasonable amount of RAM and flash memory.
- Hardware-accelerated cryptography (at least ECC on the P-256 curve, AES, SHA-256, True Random Number Generator).
- Nice-to-have: A form factor close to the physical dimensions of a security key.

We also came across an interesting presentation from Victor Lomne (NinjaLab) about the hardware used in commercial FIDO(2) security keys [20]. For example, Yubico uses NXP and Infineon microcontrollers in its products.

After thoroughly reviewing the available hardware options, we identified two candidates: Nordic Semiconductor nRF52840 and STM32H533.

OpenSK, the open-source implementation for security keys written in Rust, has chosen Nordic Semiconductor nRF52840 as its reference hardware target. nRF52840 is a SoC with a 64 MHz Cortex-M4 CPU (ARMv7-M architecture) with FPU, 1 MB Flash, 256 KB RAM, multiprotocol radio support (including Bluetooth), USB 2.0, NFC-A, and Arm TrustZone CryptoCell 310 security subsystem. There are two development boards available: nRF52840 DK (48.10 USD) and nRF52840 Dongle (10.40 USD). The nRF52840 Dongle immediately caught our attention because its form factor almost perfectly corresponds to a standard Yubico-like security key. Thus, we bought samples and evaluated them.

We did not like the quality of the documentation and supporting software libraries compared to STMicroelectronics products, so **we decided to focus our efforts on our second choice, the STM32H533 MCU on the NUCLEO-H533RE (15.95 USD) development board instead**.

STM32H533 features an Arm Cortex-M33 CPU (ARMv8-M architecture) with TrustZone, FPU, frequency up to 250 MHz, 512 KB of embedded flash memory with ECC, two banks read-while-write, 48-KB per bank with high-cycling capability (100 K cycles) for data flash, 272 KB of SRAM (80-KB SRAM2 with ECC), USB 2.0 FS and hardware-cryptography peripherals (PKA (Public Key Accelerator) with support for ECC P-256, AES, HASH, RNG) [21]. Those specs are a great fit for our use case.

Figure 4.3. nRF52840 Dongle

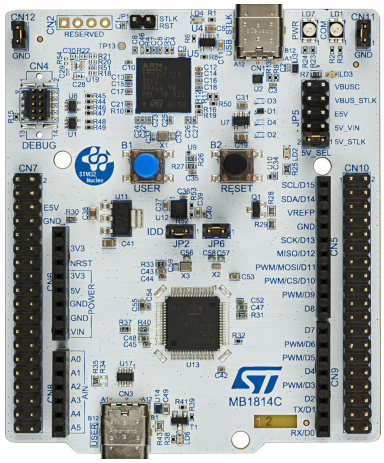

Figure 4.4. NUCLEO-H533RE

4.5 USB

The USB HID and USB layers are implemented using the TinyUSB[1] library, which support the STM32H533's USB peripheral.

There are two endpoints (IN, OUT) with interrupt transfer (64-byte packet max, poll every 5 millisecond). The descriptors (device, config, interface, endpoints, HID report) are set up according to the CTAPHID specification (11.2.8. HID device implementation) [12].

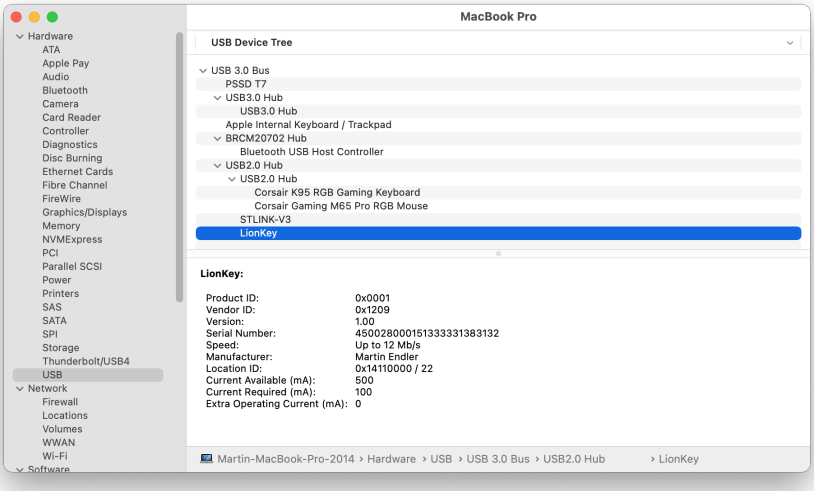

Figure 4.5. Our authenticator (LionKey) shown in the USB Device Tree Explorer in System Information on macOS 11.

The chosen Vendor ID (VID) and Product ID (VID) come from the pid.codes project. pid.codes is a registry of USB PID codes for open source hardware projects. They assign PIDs on any VID they own to any open-source hardware project needing one [22]. For the initial version of this thesis, we used their testing VID/PID `0x1209/0x0001`. In the future, we might eventually ask for our own PID.

[1] `https://github.com/hathach/tinyusb`

4.6 CBOR

We use TinyCBOR[2] for parsing (decoding) and encoding CBOR.

4.7 Arbitrary-length Strings

One challenging aspect of a CTAP2 implementation is the need to support strings of arbitrary length. We implemented an efficient zero-copy solution based on the TinyCBOR's string chunks API. When the strings need to be truncated for storage, the credentials storage submodule takes care of that. When an RP ID gets truncated, we fall back to use its SHA-256 for comparisons.

4.8 PIN/UV Auth Protocol

CTAP 2.1 defines two versions of the PIN/UV Auth Protocol. They both share a common abstract interface and only differ in their internal behavior (e.g., key derivation function, encryption, and authentication parameters). Therefore, using object-oriented programming (OOP) techniques for their implementation was a natural choice. The resulting implementation allows for easy addition of new versions without modifying the rest of the code. More importantly, the code closely matches the specification, making it easy to understand.

4.9 Cryptography

Our CTAP2 implementation requires an implementation of ECDSA and ECDH on the P-256 curve, AES-256 in CBC mode, SHA-256, HMAC [23], and HKDF [24].

We developed custom generic implementations of HMAC and HKDF from scratch. They can work with any underlying hash algorithm implementation (ether software-based or hardware-based).

In order to enable testability without the need for special hardware, we first implemented Random Number Generation (allowing custom seed for testing predictability), ECC (ECDSA and ECDH), AES, and SHA-256 using third-party software libraries optimized for use in embedded devices but compatible with standard processor architectures as well. This way we got a baseline that ran on both the host and the target hardware.

Then, we implemented the same set of functions (again using OOP) using the STM32H533 cryptography peripherals RNG, PKA (Public Key Accelerator), SHA, AES. The ability to compare the results between the software- and hardware-based implementation were crucial to debug various tricky issues.

4.10 Debugging Interface via UART

In order to allow easy debugging (and development of the USB communication stack), we implemented a simple bidirectional debugging interface using the UART peripheral. We wrote a simple write syscall implementation so that the printf statements (wrapped in custom `debug/info/error_log` macros) could work as expected. The handling of the incoming debug commands is done in the `app_debug_task()` function.

[2] `https://github.com/intel/tinycbor`

4.11 Persistent Storage

4.11.1 Requirements

There are multiple pieces of data that need to be persisted. Namely, these are:

- Whether a PIN has been set. If yes, then also:
  - The PIN hash, resp. only the first 16 bytes of the 32-byte SHA256 hash according to the spec [12].
  - The PIN length (in Unicode code points).
  - The remaining PIN attempts counter, which is initialized to 8. It decreases with each incorrect PIN entry, but it resets back to 8 after the correct PIN is entered. After **three** consecutive incorrect entries (failed attempts), a power cycle (authenticator reboot/reconnect) is required by the spec. This is done so that malicious applications running on the platform cannot block the device without user interaction [12]. It is necessary because the `authenticatorClientPIN` subcommands can be invoked by applications without any user interaction (user presence check).
- The current minimum PIN length (in Unicode code points). We store this so that we can easily add support for the optional `setMinPINLength` subcommand of the `authenticatorConfig` command and the Minimum PIN Length Extension (minPinLength) in the future.
- Credentials
  - It is necessary to store all client-side discoverable credentials (passkeys).
  - But the non-discoverable credentials do not need to be stored. Instead, during authenticator persistent state initialization (e.g., just once), we could generate and store a single symmetric encryption key and use it to encrypt (wrap) the private keys of all non-discoverable credentials as we described in the Credential Storage Modality section.
  - However, in our current implementation, we store all credentials.

4.11.2 Storage Layer Abstraction

We have implemented an abstraction, Storage API, that allows the use of different storage backends (volatile RAM memory vs. persistent flash memory). This abstraction works with **items**. An **item** is a block of data with a given **key** (`uint32_t`) and **size** (`uint32_t`). Multiple items can have the same key. Using the Storage API, one can iterate over items (possibly filtering them by key), create new items, and update or delete the existing ones. On top of that, the Storage API also includes support for a single counter (increment-only) and a method to erase the whole storage at once (atomically). CTAP2 layer uses this abstraction to store credentials and the PIN state as items.

4.11.3 Memory Storage

We have developed a universal in-memory storage layer that uses a fixed region of (volatile) memory (e.g., RAM). This layer does not depend on the hardware and can be used in the unit tests that run on the host.

4.11.4 Flash Storage

Next step was the development of the flash storage layer for the STM32H533 MCU.

STM32H533 features **512 KB** of embedded flash memory (user main memory) for storing programs and data. The flash memory is divided into two independent banks (Bank 1 and Bank 2), and it supports a read in one bank while a write operation or an erase operation is executed in the other bank (RWW – read while write). Each bank contains 256 KB of user main memory divided into 32 sectors (each sector has a size of 8 KB). The user main memory block features flash-word rows of 128 bits (16 bytes) + 9 bits of ECC per word.

Up to 8 sectors per each bank of the user main memory can be configured (via the `FLASH_EDATA1R_PRG`, resp. `FLASH_EDATA2R_PRG` registers) to support *high cycling capability* (100 K cycles [21]) for data. When configured like this, this area is protected by a robust 6-bit ECC, enabling a 16-bit read and write granularity, at the expense of having the sector size shrunk to 6 KB (max 8 * 6 = 48 KB of the high-cycling data memory per bank, 96 KB in total) [25].

In our initial proof-of-concept implementation, we reserve Bank 1 for the application code (firmware/program) and Bank 2 for the application data. This way we can leverage the RWW feature. Within Bank 2, we use the standard sector 0 (8 KB, word size = 128 bits) to store storage metadata (version and delete marker) and the actual data (the items). Furthermore, we use the high-cycling sector 0 (6 KB, word size = 32 bits) to store the signature counter values. The Storage API erase operation is implemented atomically using the delete marker and two flash sector erases. The complete flash storage implementation can be found on GitHub[3].

Our flash storage implementation is **fully functional and usable**. However, it lacks support for multiple sectors and it cannot perform compaction (free up space by re-ordering items and removing the deleted ones). Also, there are some possible edge cases we should correctly handle. We plan to address those limitations in future releases.

[3] `https://github.com/pokusew/lionkey/blob/main/targets/stm32h533/Src/flash_storage.c`

Chapter 5
Results and Evaluation

5.1 LionKey

To support adoption of our implementation by people from all over the world, we gave our implementation a name – LionKey – and we designed a logo. We also purchased the domain `lionkey.dev` (and `lion-key.dev`, `lionkey.cz`, `lion-key.cz`) and launched a website with information about the project and with documentation. We plan to gradually add more detailed documentation.

Figure 5.1. The LionKey logo.

5.2 Key Features of LionKey

In terms of WebAuthn, LionKey is a roaming authenticator with cross-platform attachment using CTAP 2.1 over USB 2.0 (CTAPHID) as the communication protocol, supporting user verification using PIN (CTAP2 ClientPIN), and capable of storing passkeys (client-side discoverable credentials).

- **Fully compliant implementation of CTAP 2.1.**
  - Implements all mandatory features.
  - Written in **C**.
  - No dynamic memory allocations.
  - Designed for use in resource-constrained environments.
  - MCU independent, easily portable, can be used as a library (see the core dir).
  - Uses TinyCBOR library for CBOR parsing and encoding.
- Runs on the NUCLEO-H533RE board with the STM32H533RET6 MCU.
  - Uses TinyUSB library for the USB and USB HID implementation.
  - Hardware-accelerated cryptography using the RNG, PKA, AES, SHA peripherals.
  - Uses STM32CubeH5 (CMSIS, HAL, LL).
- **Minimum number of external dependencies**: TinyCBOR, TinyUSB, STM32CubeH5.
- The complete source code and additional documentation is available on GitHub: `https://github.com/pokusew/lionkey`

5.3 FIDO Conformance Tools

The FIDO Alliance provides a testing tool to verify the conformance of a CTAP2 implementation. This tool is not public and it is only provided upon request. Upon our request, we were granted the access. This allowed us to further verify our implementation. As can be seen in Figure 5.2, our implementation, **LionKey**, **successfully passed all tests**, thus demonstrating **full compliance with CTAP 2.1**. With these successful test results, we could apply for the FIDO Authenticator Level 1 (L1) certification[1].

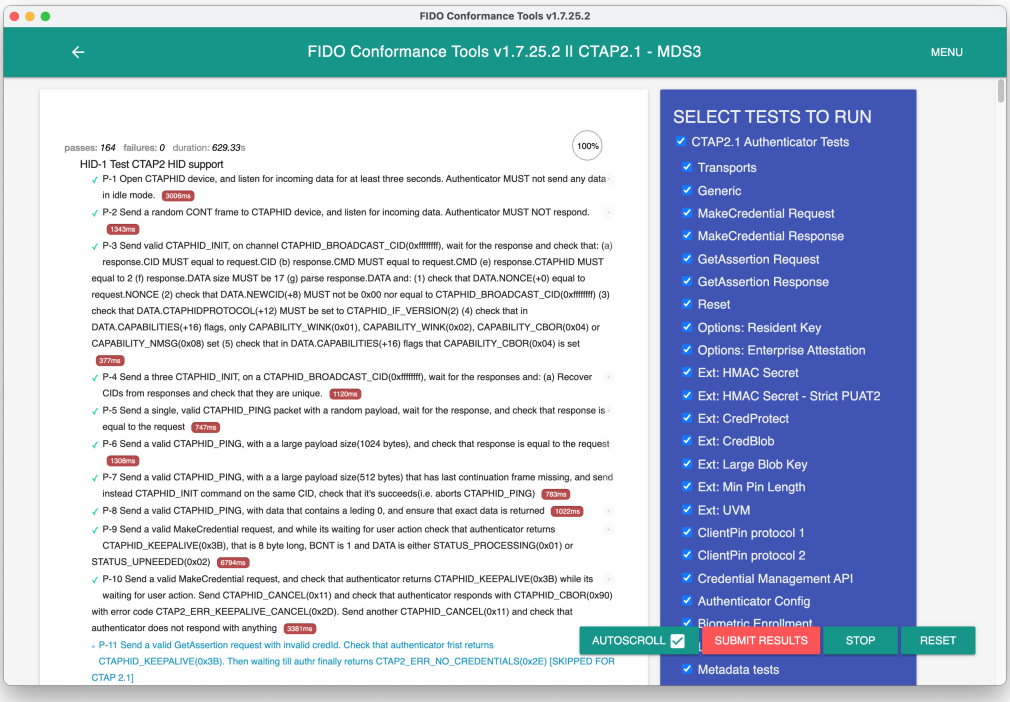

Figure 5.2. The screenshot of the FIDO Conformance Tools v1.7.25.2 application after completing the full CTAP 2.1 test suite while testing LionKey. The total duration was a little over 10 minutes. All 164 tests passed. Our implementation is fully compliant with CTAP 2.1.

5.3.1 Automation

The FIDO Conformance Tools is a desktop application (built using Electron.js and Node.js). In addition to testing FIDO CTAP 2.1 and CTAP 2.0 authenticators, it also supports testing of implementations of FIDO UF2 and FIDO UAF. Until recently, the application did not support any form of test automation. During testing, it was necessary to manually perform user presence checks each time the testing tool requested it (e.g., press a button on the authenticator and then confirm a dialog in the app). Additionally, the authenticator needed to be manually reconnected to allow the

[1] More infromation about the FIDO certification process can be found at `https://fidoalliance.org/certification/functional-certification/` and `https://fidoalliance.org/certification/authenticator-certification-levels/`

`authenticatorReset` command, as CTAP2 authenticators *without display* can only be reset within the first 10 seconds after powering on.[2]

Starting with the version v1.7.25, the FIDO Conformance Tools app provides an option to enable test automation. When enabled, this feature minimizes user interaction and reduces manual input when running the tests. Instead of relying on pop-up dialogs and manual user interaction, the app sends HTTP POST requests to trigger the appropriate action (authenticator power cycle, user presence test, etc.). "The idea is that these actions will be handled by an additional peripheral device instead of human action." [26].

Therefore, to streamline the testing, we have developed a simple Node.js test automation server that implements the FIDO Conformance Tools Automation API. This server connects to the authenticator via the serial port and utilizes our UART Debugging Interface to send the appropriate commands to the authenticator whenever it receives an automation HTTP POST request. Additionally, it forwards all standard input and output to and from the serial port, allowing it to function as a serial console. The automation server script can be found in the `tools` directory of our repository.[3]

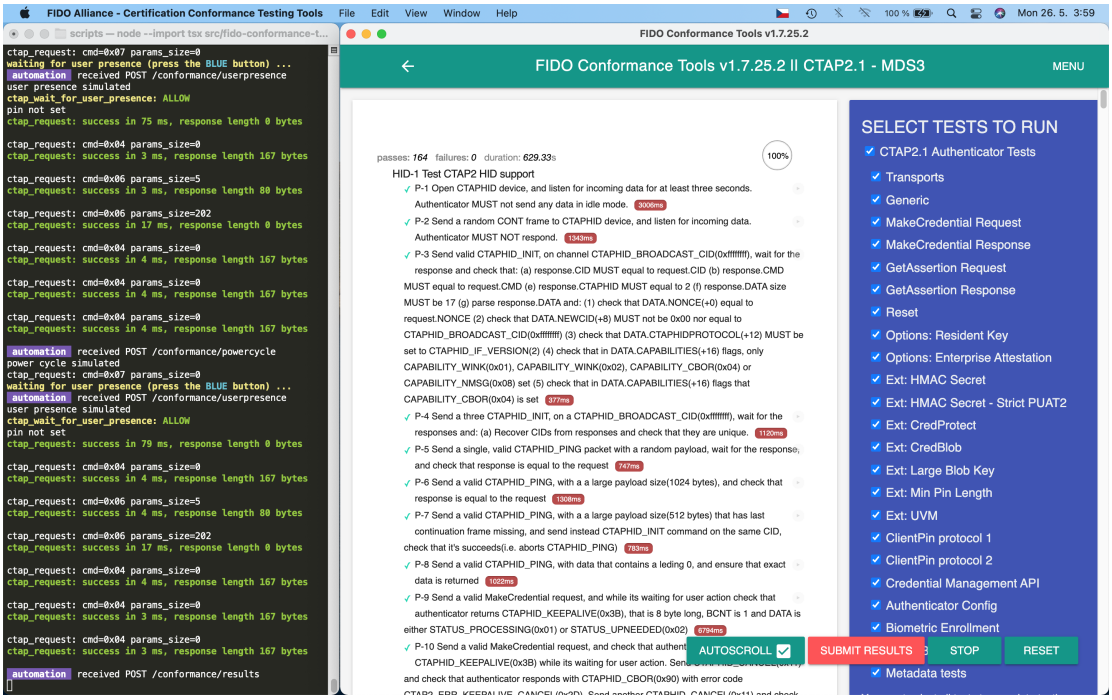

Figure 5.3. The screenshot of the FIDO Conformance Tools v1.7.25.2 application with the output of our test automation server on the left side.

5.3.2 Authenticator Metadata Statement

A FIDO Authenticator Metadata statement is a JSON document that contains information (characteristics and capabilities) about a specific FIDO authenticator. [27]

The FIDO Alliance operates the FIDO Metadata Service, which provides a method for relying parties to obtain FIDO Metadata statements in a way that provides a chain of

[2] This measure is included in the spec to prevent accidental/malicious triggering of the reset, because when the authenticator does not have a display, the user cannot know what action they are actually confirming (via the user presence check).

[3] `https://github.com/pokusew/lionkey`

trust and makes it possible to validate attestation signatures generated by a particular authenticator. Relying parties can use the authenticator metadata to make policy decisions about authenticators. [28]

The authenticator metadata statement is also required by the FIDO Conformance Tools testing app when performing the authenticator tests (CTAP 2.1 or CTAP 2.0).

The LionKey Authenticator Metadata Statement can be found in the `tools` directory of our repository.[4]

One part of the metadata statement is also a snapshot of the response to the `authenticatorGetInfo` CTAP2 command.

Below, we provide the `authenticatorGetInfo` response of LionKey when it is in its initial (factory) state. Note that response is dynamic and it changes based on the current authenticator's state. For example, the `clientPin: false` part indicates the PIN is currently not set. Once the PIN is set, it will change to `clientPin: true`. The absence of the `clientPin` option would indicate that the CTAP2 ClientPIN feature is not supported.

```
{
   "versions": [
      "FIDO_2_1"
   ],
   "extensions": [
      "credProtect",
      "hmac-secret"
   ],
   "aaguid": "a8a147326b7d120dfb917356bc189803",
   "options": {
      "rk": true,
      "up": true,
      "plat": false,
      "credMgmt": true,
      "clientPin": false,
      "pinUvAuthToken": true,
      "makeCredUvNotRqd": true
   },
   "pinUvAuthProtocols": [
      2,
      1
   ],
   "maxCredentialCountInList": 128,
   "maxCredentialIdLength": 128,
   "algorithms": [
      {
         "alg": -7,
         "type": "public-key"
      }
   ],
   "minPINLength": 4
}
```

[4] https://github.com/pokusew/lionkey

5.4 Use on Real WebAuthn-enabled Websites

We tested our security key with real WebAuthn-enabled websites. In all cases it worked flawlessly. In particular, we verified that it can be used for authentication on **Google** and **GitHub**. This further confirms the correctness of our implementation.

5.4.1 CTU FEE

At CTU FEE, WebAuthn (passkeys) authentication has been recently implemented. Of course, we tested LionKey with that as well. It worked perfectly. Both the full registration and authentication flows are depicted in the following screenshots.

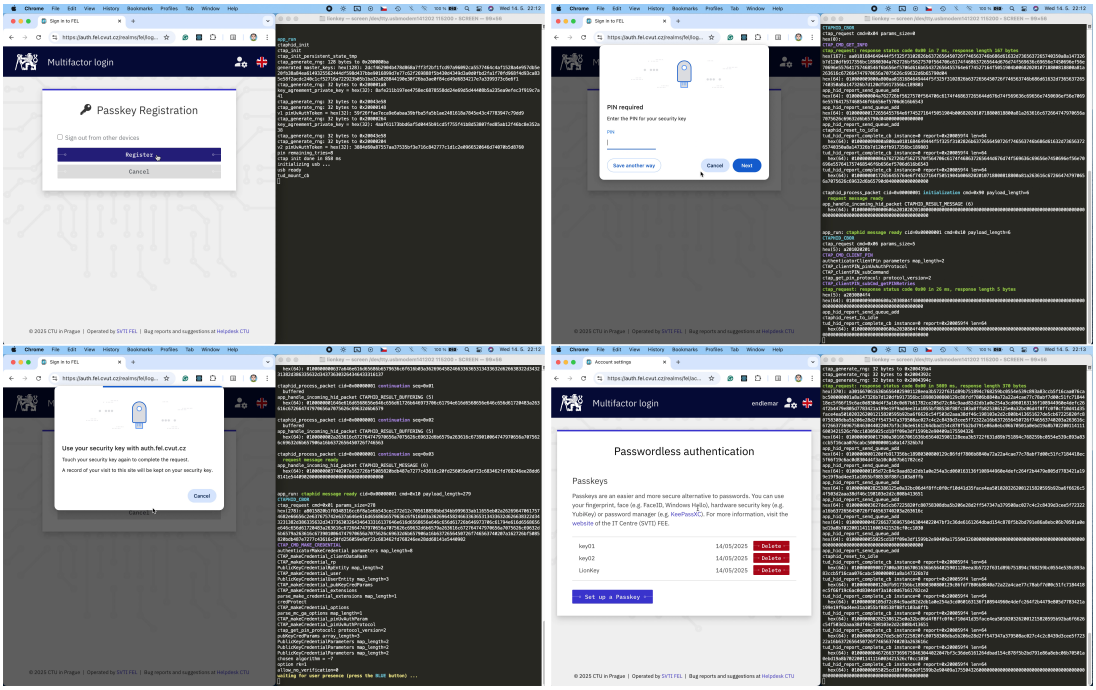

Figure 5.4. The full WebAuthn flow of a creating a new passkey with LionKey on auth.fel.cvut.cz. **1.** (top left image) Adding a new passkey. **2.** (top right image) User verification (LionKey supports only PIN) is required for passkeys (aka client-side discoverable credentials). **3.** (bottom left) User presence check – The user consents to the action of creating a new credential. **4.** (bottom right) The UI of the auth.fel.cvut.cz website confirms that the passkey was successfully registered.

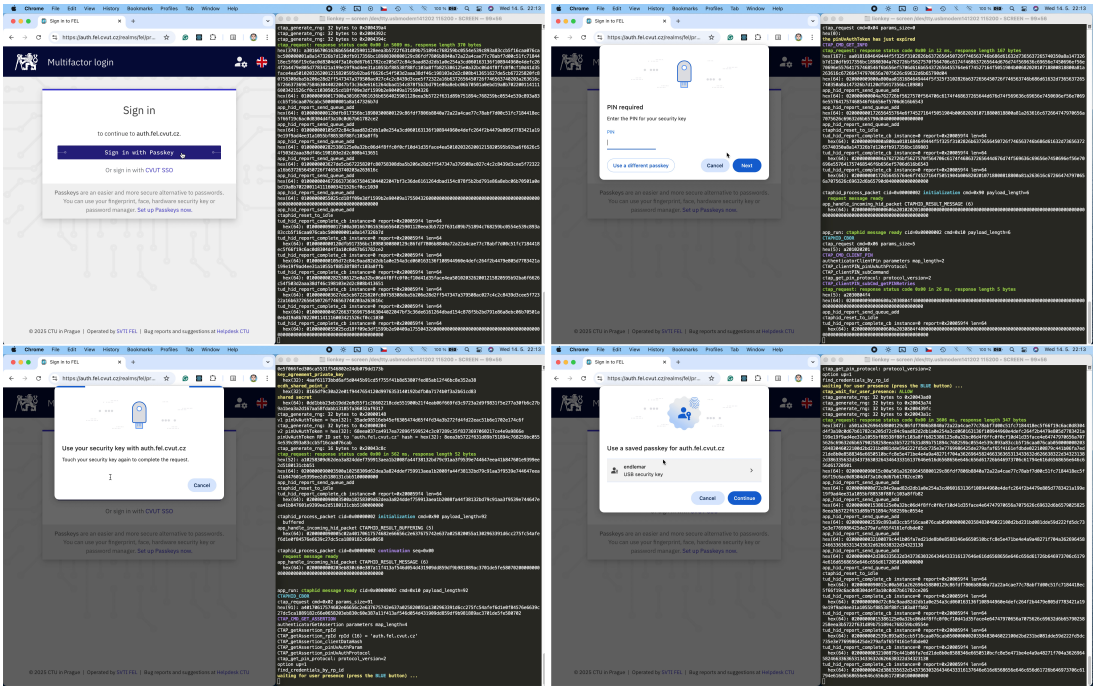

Figure 5.5. The full WebAuthn flow of a using LionKey with passkeys for authentication on auth.fel.cvut.cz. **1.** (top left image) Authenticating with an existing passkey. **2.** (top right image) User verification (LionKey supports only PIN) is required for passkeys (aka client-side discoverable credentials). **3.** (bottom left) User presence check – The user consents to the action of authenticating with an existing passkey (generating assertions for all passkeys stored for the RP ID auth.fel.cvut.cz). **4.** (bottom right) The browser UI asks the user to choose one of the returned passkeys from the authenticator.

Chapter 6
Conclusion

In this thesis, **we successfully implemented** a fully functional FIDO2 USB hardware external authenticator (also called FIDO2 USB security key).

First, we outlined the objective of our work and its motivation. Then, we established the necessary theoretical foundation by describing the Web Authentication and CTAP2 standards, which are together known as FIDO2. We also reviewed existing projects related to our goal.

Then, we proceeded to the actual implementation, which represents the main contribution of this thesis. We documented our decisions and the most important parts of the implementation. All code is versioned using Git and is publicly **available online on GitHub**[1]. The repository includes build and run instructions. We employed software development best practices, including unit testing and CI/CD pipeline.

Finally, **we tested** our authenticator **with real websites** with different configurations. In all cases, **it worked great**, including with Google Account and GitHub. Furthermore, we tested our authenticator using the official FIDO Conformance Tools testing application. Our implementation passed the entire CTAP 2.1 test suite, demonstrating the quality of our work.

To support adoption of our implementation by people from all over the world, we gave our implementation a name – LionKey – and we designed a logo. We also purchased the domain `lionkey.dev` and launched a website with information about the project and with documentation. We plan to gradually add more detailed documentation.

Figure 6.1. The LionKey logo.

[1] `https://github.com/pokusew/lionkey`

References

[1] Verizon. *Data Breach Investigations Report*. 2020.
https://www.verizon.com/business/en-gb/resources/reports/2020-data-breach-investigations-report.pdf.
Additional information at
https://www.verizon.com/business/resources/reports/dbir/.

[2] Bitwarden. *Bitwarden World Password Day Survey*. 2024.
https://bitwarden.com/resources/world-password-day/#2024-world-password-day-results.

[3] FIDO Alliance. *Overview*.
https://fidoalliance.org/overview/.

[4] MDN Web Docs. *Web Authentication API – Browser compatibility*.
https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API#browser_compatibility.

[5] Can I use… . *Web Authentication API*.
https://caniuse.com/webauthn.

[6] passkeys.dev . *Device Support*.
https://passkeys.dev/device-support/.

[7] FIDO Alliance. *Member Companies & Organizations*.
https://fidoalliance.org/members/.

[8] FIDO Alliance. *Apple, Google and Microsoft Commit to Expanded Support for FIDO Standard to Accelerate Availability of Passwordless Sign-Ins*. May 5, 2022.
https://fidoalliance.org/apple-google-and-microsoft-commit-to-expanded-support-for-fido-standard-to-accelerate-availability-of-passwordless-sign-ins/.

[9] FIDO Alliance. *Authenticator Certification Levels*.
https://fidoalliance.org/certification/authenticator-certification-levels/.

[10] W3C. *Web Authentication: An API for accessing Public Key Credentials – Level 2*. W3C Recommendation. April 8, 2021.
https://www.w3.org/TR/webauthn-2/.

[11] W3C. *Web Authentication: An API for accessing Public Key Credentials – Level 3*. W3C Working Draft. January 27, 2025.
https://www.w3.org/TR/webauthn-3/.

[12] FIDO Alliance. *Client to Authenticator Protocol (CTAP). CTAP 2.1*. Proposed Standard. June 21, 2022.
https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-errata-20220621.html.
.

[13] FIDO Alliance. *Client to Authenticator Protocol (CTAP). CTAP 2.2*. Proposed Standard. February 28, 2025.
https://fidoalliance.org/specs/fido-v2.2-ps-20250228/fido-client-to-authenticator-protocol-v2.2-ps-20250228.html.

[14] Wikipedia. *Public-key cryptography*.
https://en.wikipedia.org/wiki/Public-key_cryptography.

[15] Wikipedia. *RSA (cryptosystem)*.
https://en.wikipedia.org/wiki/RSA_(cryptosystem).

[16] Wikipedia. *Elliptic-curve cryptography*.
https://en.wikipedia.org/wiki/RSA_(cryptosystem).

[17] USB Implementers Forum. *HID Usage Tables 1.6*. January 30, 2025.
https://usb.org/document-library/hid-usage-tables-16.

[18] Matěj Borský. *FIDO2 Authentication Simulator*. Bachelor's Thesis, Czech Technical University in Prague, Faculty of Information Technology. May 2022.
https://dspace.cvut.cz/handle/10467/101916.

[19] Martin Kolárik. *FIDO2 KeePass Plugin*. Bachelor's Thesis, Czech Technical University in Prague, Faculty of Information Technology. June 2020.
https://dspace.cvut.cz/handle/10467/88264.

[20] Victor Lomne (NinjaLab). *An Overview of the Security of Some Hardware FIDO(2) Tokens*. October 28, 2022.
https://hardwear.io/netherlands-2022/presentation/security-of-Hardware-FIDO(2)-tokens.pdf.

[21] STMicroelectronics. *STM32H533xx*. Datasheet DS14539. November 20, 2024.
https://www.st.com/resource/en/datasheet/stm32h533re.pdf.

[22] pid.codes . *Free USB VID and PID codes for open-source projects*.
https://pid.codes/.

[23] Hugo Krawczyk, and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010.
https://datatracker.ietf.org/doc/html/rfc5869.

[24] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. February 1997.
https://datatracker.ietf.org/doc/html/rfc2104.

[25] STMicroelectronics. *STM32H523/33xx, STM32H562/63xx, and STM32H573xx Arm®-based 32-bit MCUs*. Reference manual RM0481. April 24, 2025.
https://www.st.com/resource/en/reference_manual/rm0481-stm32h52333xx-stm32h56263xx-and-stm32h573xx-armbased-32bit-mcus-stmicroelectronics.pdf.

[26] FIDO Conformance Test Tools Resources. *Test Automation*. May 7, 2025.
https://github.com/fido-alliance/conformance-test-tools-resources/blob/main/docs/FIDO2/Automation.md.

[27] FIDO Alliance. *FIDO Metadata Statement 3.0*. Proposed Standard. May 18, 2021.
https://fidoalliance.org/specs/mds/fido-metadata-statement-v3.0-ps-20210518.html.

[28] FIDO Alliance. *FIDO Metadata Service 3.0*. Proposed Standard. May 18, 2021.
https://fidoalliance.org/specs/mds/fido-metadata-service-v3.0-ps-20210518.html.

Appendix A
Glossary

AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation One
CBOR	Concise Binary Object Representation
CMSIS	Common Microcontroller Software Interface Standard
CTAP	Client to Authenticator Protocol
CTU FEE	Czech Technical University in Prague, Faculty of Electrical Engineering
DH	Diffie-Hellman
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography (in the context of cryptography) Error Correction Code (in the context of FLASH)
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
FIDO	Fast IDentity Online Alliance
HAL	Hardware Abstraction Layer
HID	USB Human Interface Devices
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KDF	Key Derivation Function
LL	STM32 Low Layer (LL) drivers
MCU	microcontroller unit
MFA	multi-factor authentication
OOP	Object-Oriented Programming
OS	Operating System
RSA	The RSA (Rivest–Shamir–Adleman) cryptosystem
SHA-2	Secure Hash Algorithm 2 (includes SHA-256)
UI	User Interface
USB	Universal Serial Bus
2FA	second-factor authentication